Balancing the Quality and Cost of Updating Dependencies

Damien Jaime
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
SAP France S.A.
F-92300, Levallois-Perret, France
damien.jaime@lip6.fr

Pascal Poizat
Sorbonne Université, CNRS, LIP6
F-75005, Paris, France
Université Paris Nanterre
F-92000, Nanterre, France
pascal.poizat@lip6.fr

Joyce El Haddad
Université Paris Dauphine-PSL, CNRS, LAMSADE
F-75016, Paris, France
joyce.elhaddad@lamsade.dauphine.fr

Thomas Degueule
Univ. Bordeaux, CNRS, Bordeaux INP, LaBRI
UMR 5800, France
thomas.degueule@labri.fr

ABSTRACT

Keeping dependencies up-to-date is a crucial software maintenance task that requires significant effort. Developers must choose which dependencies to update, select appropriate target versions, and minimize the impact of updates in terms of breaking changes and incompatibilities. Several factors influence the choice of a new dependency version, including its freshness, popularity, absence of vulnerabilities, and compatibility.

In this paper, we propose to formulate the dependency update problem as a multi-objective optimization problem. This approach allows for the update of dependencies with a global perspective, considering all direct and indirect dependencies. It also enables developers to specify their preferences regarding the quality factors they want to maximize and the costs of updating they want to minimize. The update problem is encoded as a linear program whose solution is an optimal update strategy that aligns with developer priorities and minimizes incompatibilities.

We evaluated our approach using a dataset of 107 well-tested open-source Java projects using various configurations that reflect real-world update scenarios, and considered three quality metrics: dependency freshness, a time-window popularity measure, and a vulnerability score related to CVEs. Our findings indicate that our approach generates updates that compile and pass tests as well as naive approaches typically implemented in dependency bots. Furthermore, our approach can be up to two orders of magnitude better in terms of freshness. By considering a more comprehensive concept of quality debt, which takes into account freshness, popularity, and vulnerabilities, our approach is able to reduce quality debt while maintaining reasonable memory and time consumption.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories*; *Software maintenance tools*; *Software evolution.*

KEYWORDS

software maintenance, dependency graph, dependency update

ASE'24, October 27 – November 1, Sacramento, CA, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Damien Jaime, Pascal Poizat, Joyce El Haddad, and Thomas Degueule. 2024. Balancing the Quality and Cost of Updating Dependencies. In *Proceedings of 39th IEEE/ACM International Conference on Automated Software Engineering (ASE'24).* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Leveraging the time-honored principles of modularity and reuse, modern software systems development typically entails using external *software libraries*. Instead of creating new systems from scratch, developers incorporate libraries that provide the desired functionalities into their projects. These libraries expose their features through Application Programming Interfaces (APIs), which dictate the interactions between client projects and libraries. The way dependencies are managed varies across different software ecosystems [9]. Typically, it involves using a package manager or build system to automatically retrieve specific versions of dependencies from remote software repositories—along with their own (transitive) dependencies—in order to build a so-called *dependency graph* [26]. For example, JavaScript and TypeScript developers can use npm or Yarn to fetch dependencies from the npm Registry, while Java developers can use Maven or Gradle to retrieve dependencies from the Maven Central repository.

Libraries continuously evolve to incorporate new features, bug fixes, security patches, refactorings, *etc.* [1, 4]. Clients must stay up-to-date with the libraries they use to benefit from these improvements and to avoid technical lag and the associated technical debt [5, 11, 27]. However, when a library evolves, it may introduce changes that break the contract previously established with its clients, resulting in syntactic and semantic errors [33]. Developers are thus faced with the challenge of maximizing the freshness and quality of their dependencies while minimizing the costs associated with updating them. This challenge is further complicated by the nature of dependency graphs: updating a single dependency can cause a snowball effect and result in incompatibilities with other indirect dependencies. As a result, clients sometimes hesitate to update their dependencies, raising security concerns [20] and making future updates even more difficult [23].

The problem of assisting developers in updating their dependencies has therefore attracted significant interest. This includes efforts to identify client code that is affected by breaking changes [23], automatically migrate client code [34], or find versions that minimize the impact on the dependency graph. UPCY [8], in particular, is a novel approach to dependency updating that takes a library and a target version as input to construct a migration plan that minimizes the number of breaking changes within the graph induced by version updates. While this approach is particularly suitable for updating a single dependency to a specific version (*e.g.*, to avoid a particular vulnerability, as exemplified by the recent log4shell mayhem), it is much less suitable for updating the entire dependency graph at once. Besides, breaking changes may not accurately reflect the actual impact an update has, as it has been empirically shown that most breaking releases do not impact client projects in practice [18, 23].

Migrating to the latest available version of each dependency may not always be the optimal choice. Developers must juggle various criteria to find a satisfactory solution, ranging from ensuring license consistency across projects [30] to minimizing security vulnerabilities [20] and easing the migration process [23]. In this paper, we propose to model the problem of updating a dependency graph as a custom multi-objective optimization problem. We formulate the optimization problem as a linear programming problem on a project-rooted extended dependency graph. Our approach is generic with respect to the considered quality and cost metrics. As different developers prioritize these criteria differently, our multi-objective problem incorporates weights for each, hence supporting updates tailored to organizational rules or individual developers' preferences. While other criteria can be considered, we focus in our experimental evaluation on the joint use of three quality metrics: dependency freshness (to minimize the cost of future updates), a time-window popularity measure (as a proxy for community support), and a vulnerability score based on CVEs (as a proxy for security concerns). For the cost of change, we estimate the impact breaking changes introduced in a release have on the project.

We develop a new tool, GoblinUpdater, that automatically proposes an update plan from developer-defined preferences. GoblinUpdater targets the Java programming language and the Maven ecosystem and leverages the Maven Dependency Graph [2] and the enrichment capabilities offered by Goblin [15] to incorporate quality and cost metrics into dependency graphs. GoblinUpdater also leverages Maracas to compute the impact of breaking changes on client code [23]. Our experiments evaluate the correctness, effectiveness, and scalability of GoblinUpdater on a dataset of 107 well-tested open-source Maven projects. We show that GoblinUpdater outperforms naive approaches while maintaining reasonable memory and time consumption.

The remainder of this paper is organized as follows. Section 2 introduces the problem of balancing the quality and cost of dependency updates and the existing approaches. Section 3 gives a general overview of our approach, Section 4 discusses the construction of project-rooted dependency graphs, and Section 5 the encoding of the problem as a linear program. Section 6 evaluates the benefits of our approach on various update configurations. Section 7 presents the threats to validity. Finally, Section 8 discusses some lessons learned in the use of linear programming for dependency update and Section 9 concludes the paper.

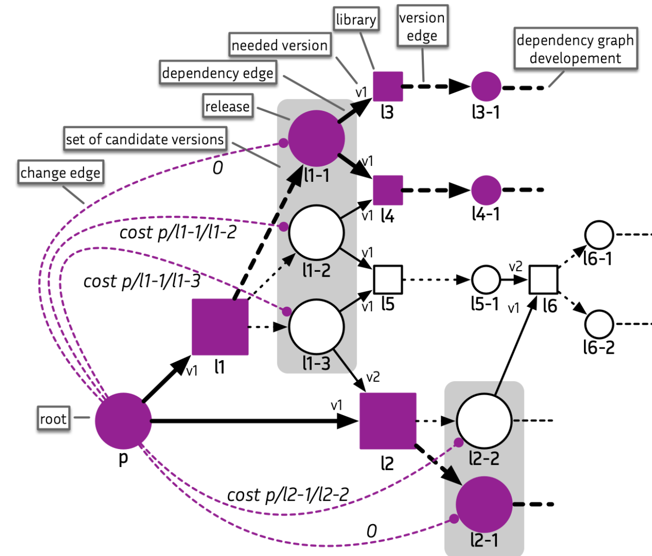


Figure 1: An extended dependency graph rooted in p. This dependency graph for p (purple/dark squares and circles) incorporates all alternative versions of p's direct dependencies (grey rounded boxes) and change edges with associated costs. The direct and indirect dependencies of these alternative versions are depicted with white squares and circles. The current dependencies of p are {l1-1, l2-1, l3-1, l4-1}. Alternative dependency graphs for p are {l1-1, l2-2, l3-1, l4-1, l6-1}, {l1-1, l2-2, l3-1, l4-1, l6-2}, {l1-2, l2-1, l4-1, l5-1, l6-1}, {l1-2, l2-1, l4-1, l5-1, l6-2}, {l1-2, l2-2, l4-1, l5-1, l6-1}, {l1-2, l2-2, l4-1, l5-1, l6-2}, {l1-3, l2-1, l5-1, l6-1}, {l1-3, l2-1, l5-1, l6-2}, {l1-3, l2-2, l5-1, l6-1}, and {l1-3, l2-2, l5-1, l6-2} (changes are underlined).

2 PROBLEM STATEMENT & RELATED WORK

To illustrate the problem of balancing the quality and cost of updating dependencies, consider the dependency graph G depicted in Figure 1. The figure depicts a simple root project p that declares two direct dependencies towards libraries l1 (in version l1-1) and l2 (in version l2-1), and inherits indirect dependencies towards libraries l3–l4. Each library offers a set of releases that act as candidates for replacing existing dependency versions (l1, for instance, offers releases l1-1, l1-2, and l1-3, with l-3 the most recent release). Migrating from one library version to another incurs a change cost, depicted with dotted arrows in Figure 1.

Updating the dependencies of project p entails finding a subgraph G' of G that satisfies ecosystem-specific well-formedness constraints (*e.g.*, one can only depend directly on a single version of a given library), maximizes the quality of each dependency in the graph rooted in p, and minimizes the cost of migrating towards new versions. A given solution must specify the version of each dependency, whether direct or transitive, to ensure that no version is left open for the dependency resolver to pick arbitrarily. The goal is to find an optimal solution with respect to specific user-defined quality and cost preferences. Even in the simple example of Figure 1, combining every candidate version of each library yields ten candidate solutions G'. In real-world settings with dozens or hundreds of dependencies, the number of possible solutions quickly becomes unmanageable with naive algorithms and approaches.

Consider the simple case where a developer wants to maximize the freshness of their dependencies while minimizing the cost of migrating to these versions. Updating would mean choosing {l1-3, l2-2, l5-1, l6-2}—assuming that these do not incur significant costs in terms of breaking changes. Another developer may opt to maximize the freshness of dependencies and minimize the presence of vulnerabilities, regardless of the cost of migrating towards this new configuration. A successful approach should enable developers to (i) precisely express their preferences regarding quality and cost and (ii) infer an optimal solution wrt. these preferences in reasonable time.

Several tools have been developed to assist developers in managing their dependencies (*e.g.*, Dependabot [12], Renovate [24], and Greenkeeper [25]). These tools monitor the release of new dependencies and automatically create pull requests submitted for approval to project maintainers. They operate at the level of individual releases and do not aim to find a global solution that satisfies user-defined criteria. UPCY [8], on the other hand, is a novel approach to dependency update that takes as input one library to update and one target version to construct a migration plan that minimizes the number of breaking changes induced by version updates across the entire dependency graph. While this approach is particularly suitable for updating a single dependency to a specific version (*e.g.*, to avoid a particular vulnerability), it is much less suitable for updating the entire dependency graph at once. Besides, the number of breaking changes is a poor indicator of the real impact an update has, as it has been shown empirically that most breaking releases do not impact client projects in practice [18, 23]. In contrast, our approach considers the *impact* of breaking changes on the root project p and allows for updating all dependencies of a project at once. Hejderup et al. studied to which extent test suites of client projects can detect regression caused by dependency updates and found that tests can only detect 47% of artificial faults injected in direct dependencies and 35% of those injected in transitive ones [13]. They advocate that a successful approach to dependency update should incorporate static analysis to compensate the inadequacy of tests. Our approach follows that path by incorporating the precise cost of dependency updates directly within the dependency graph.

3 APPROACH OVERVIEW

In this section, we give an overview of our approach for updating a project's dependencies. Our approach is depicted in Algorithm 1. It takes as input :

- the project of interest, both its POM file (to retrieve direct dependencies) and its source code (to compute the cost of change),
- a non empty set Q of user-chosen quality metrics, to which we add a specific cost of change metric (`cost`), yielding a set Q^+,
- a function w associating to each q in Q^+ a user-defined weight in $[0, 1]$, with w_q denoting the weight for q. We further require that $\sum_{q \in Q} w_q = 1$ (the sum of weights for quality metrics is 1). Note that this means that $\sum_{q \in Q^+} w_q$ is in $[1, 2]$.

The algorithm consist of two main steps: a preparatory one, and the resolution of dependency update using linear programming.

Preparation. In the first step, we start (line 1) by constructing a dependency graph called the *rooted dependency graph* (rDG) from p's direct dependencies. In addition to containing all direct and indirect dependencies for p, this graph also contains additional libraries and releases corresponding to a *potential for update*. This information is taken from the whole dependency graph hosted in Maven Central. Values for the quality metrics of interest for the user are also computed and associated to each release in the rDG. Then (line 2), the rDG is extended into a *rooted extended dependency graph* (rEDG) with information related to the cost of change when switching from a library version to another one, based on the practical use of the library by project p.

Problem solving. In the second step, we encode the update problem as a linear program that uses the rEDG, the quality metrics and cost of change values, and the weight function to formulate the problem. The objective is to determine the *optimal set of dependency updates* that balance improvement in quality against cost of change. Since all quantitative information (*i.e.*, values for quality metrics and change cost) can vary significantly in terms of scale, we have to perform normalization first (line 3) in order to ensure a consistent basis for comparison when using linear programming. Then the encoding itself can be achieved (line 4). Finally, we use a linear programming solver (line 5) to find the optimal solution (which always exists, being in the worse case p's actual set of dependencies). The solution indicates which part of the rEDG should be taken into consideration to update p's dependencies (line 6).

4 MODELS & GRAPH CONSTRUCTION

4.1 Models

Dependency graphs. Our first model is used to represent dependencies and versioning between libraries and their releases. Such models can be retrieved from software ecosystems to address ecosystem wide research questions and support software related maintenance processes. Our model is a formalization of the graph database used by Goblin [15].

Definition 1 (Dependency Graph). *A Dependency Graph (DG) G, is a tuple $(N_L, N_R, E_D, E_V, \mathrm{req})$ where N_L is a set of library nodes, N_R is a set of release nodes, $E_D \subseteq N_R \times N_L$ is the dependency relation (edges), $E_V \subseteq N_L \times N_R$ is the version relation (edges), and req is a version constraint function associating to each edge in E_D a version in a set Ver denoting semantic versions. We also define $N = N_L \cup N_R$ and $E = E_D \cup E_V$.*

Algorithm 1 Update project dependencies

Inputs: p project (code and POM file), $Q^+ = Q \cup \{\texttt{cost}\}$ set of metric values, w weight function
Output: p' an update of p

```
1: G ← computeRDG(p.dependencies, Q)
2: G ← extendRDG(p, G)
3: G ← normalize(G, Q+)
4: program ← generateLinearProgram(G, Q+, w)
5: solution ← solve(program)
6: p′ ← update(p, solution)
7: return p′
```

An edge $e = (\mathsf{r}, \mathsf{l})$ in E_D denotes a dependency relation between release r and library l, with required version being req(e). An experiment we have done has shown that on Maven Central only approximately 1% of dependency relation use range version requirements (*e.g.*, `[1.0, 2.0)`). Hence, we suppose here that Ver strictly corresponds to semantics versions and not ranges. An edge (l, r) in E_V denotes a version relation between l and r meaning that r is a version of l.

Rooted dependency graphs. We say that a DG G is *rooted* when one has a distinct node p in N_R and that G contains only nodes and edges that are reachable from p. In this paper we only use rooted DGs, or rDGs, since our objective is to update the dependencies of a project of interest, which acts as the root for the graph. There are *different strategies to compute rDGs* when it comes to the versions of libraries. For libraries that are direct dependencies of the root we may for example keep all versions, only versions at most as old as the one required by the root, or only non patch versions. For libraries that are not direct dependencies of the root (*i.e.*, they are indirect dependencies) the same kind of strategies apply, in addition to keeping only into account versions that are required. The choice of a strategy to compute an rDG has implication on its size, on possibilities for update and on time/memory required for the computation of the best update plan.

Extended dependency graphs. In order to support dependency update, we extend rDGs with a new kind of edge denoting the cost of changing from a version to another one. This is done using change edges and a cost function associated to them.

Definition 2 (Extended Dependency Graph). *An Extended Dependency Graph (EDG) G, is a tuple* $(N_L, N_R, E_D, E_V, E_C, \mathrm{req}, \mathrm{cost})$ *such that* $(N_L, N_R, E_D, E_V, \mathrm{req})$ *is a DG and where* $E_C \subseteq N_R \times N_R$ *is the change relation (edges), and* cost *is a cost function associating to each edge in* E_C *a cost in some abstract set Cost (a measure of change debt is typically used here, see in the sequel). Further, we require that one can have an edge* $(\mathsf{r}_1, \mathsf{r}_2)$ *in* E_C *only if one has an edge* $(\mathsf{r}_1, \mathsf{l})$ *in* E_D *and an edge* $(\mathsf{l}, \mathsf{r}_2)$ *in* E_V.

Rooted extended dependency graphs. As for DGs and rooted DGs, we have EDGs and rooted EDGs, or rEDGs. As for the computation of rDGs, the computation of change edges is a matter of *strategy*. One strategy is to compute the maximal set of possible change edges (*i.e.*, all that fulfil the requirement in Definition 2). Another one is to compute only change edges outgoing from the root.

Illustration. An example of an rEDG is given in Fig. 1. There, the strategy for the computation of the rDG is all versions for direct dependencies. This also corresponds, here, to all more recent versions. For indirect dependencies, the strategy here is only used versions, see, *e.g.*, the two versions used for l6. So, if l3, l4, or l5 has more than one version, only one is present in the graph. Then for the computation of the change edges to get an rEDG, the strategy is all possible change edges for the root and none for the other nodes. Other combination of strategies would have yield other rEDGs.

4.2 Rooted dependency graph construction

To construct the rDG for a project p, we rely on features provided by Goblin [15]: the whole Maven Central dependency graph stored in a graph database, possibility to retrieve sub-graphs of it using predefined REST routes or Cypher [22] queries, and computation and adding of values to the nodes and edges of the graphs.

Regarding the added values required in our experiments (Sect. 6), we reused Goblin's "CVE" and "Freshness" added values associated to release nodes. For the later we reused as-is the data computed by Goblin. For CVE we perform a post-treatment since Goblin only gives us the list of CVEs that impact a release. To make this usable for updating, we computed the number of CVE in each of four criticality categories and aggregated them using coefficients from the Fibonacci suite. We also extended Goblin with a new added value to compute the specific popularity metric we rely on.

As far as the rDG structure itself is concerned, we also had to extend Goblin to make it amenable to our update problem. For this we defined a new route with parameters corresponding to possible strategies for rDG extraction.

4.3 Rooted extended dependency graph construction

Once we have an rDG, we must add to it information to integrate the cost of change in the update process, *i.e.*, add the edges in E_C and the values in cost. The first thing is to decide which change edges we want. As discussed before this is a matter of strategy. We can do this only for the root. This is what is illustrated in Fig. 1. We can also do this each time there is a library in the rDG with more than one version. For example, on Fig. 1 this would mean adding six more change edges: (l2-2, l6-1), (l2-2, l6-2), (l1-3, l2-1), (l1-3, l2-2), (l5-1, l6-1), and (l5-1, l6-2).

The idea behind the computation of the cost of change is to accept possible breaking changes in exchange for a better dependency quality. The cost of change is computed, for each change edge as follows. Take a release r that depends on a library l, and actually (we use req to get this information) uses version r_i of l. Suppose we want to compute the cost on the change edge between r and another version of l, say r_j. For this we call Maracas by giving it: the jar file of r_i, the jar file of r_j and the code of r. Maracas then computes all breaking changes (*e.g.*, changes in method names or arguments, class names, accessibility) between what is used by r in r_i and what will be available in r_j. From this information we get the number of all such breaking changes as the cost of change. An optimisation we apply is not to use Maracas when we compare a release to itself (the cost is always zero in this case).

It should be noted that in the case one accepts possible breaking changes in indirect dependencies (*e.g.*, for l5-1 (resp. l2-2) using l6-1 (resp. l6-2) instead of l6-2 (resp. l6-1) one cannot use Maracas to compute a cost of change. Instead, we use the Japicmp tool [21]. Since this tool can only compare two releases, and not the use of them as Maracas did, the cost is in this case is a pessimistic metric.

5 LINEAR PROGRAMMING FOR DEPENDENCY UPDATE

In this section, we present how dependency update can be encoded as a linear program that maximizes the quality of dependencies while minimizing the cost of change.

In a linear program [6] three elements are required: a set of *decision variables*, a set of *constraints*, and an *objective function*, where both the constraints and the objective function must be linear. The output of a linear program is the optimal value of the objective function (maximum or minimum) and the corresponding values of the decision variables that achieve this optimum.

5.1 Graph-based decision variables

We use the following decision variables:

- for each library l in N_L, a binary variable $\mathrm{v}_{\mathrm{l}}^{\mathrm{lib}}$ representing whether l is present (equals 1) or not (equals 0) in the solution,
- for each release r in N_R, a binary variable $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$ representing whether r is present or not in the solution,
- for each change edge between release nodes r and r' in E_C, a binary variable $\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}}$ representing whether the edge is present or not in the solution.

These decision variables are used in the sequel to express constraints that an update solution has to fulfil.

5.2 Conditions & constraints for a valid update

Several conditions are required for an update solution to be correct (whether optimal or not):

- for releases, (a) the root is present, (b) if a release (including root) is present then all its dependencies (libraries) are present, and (c) if a release (but for root) is present then the library it is a version of, is present.
- for libraries, if a library is present, then (d) exactly one of its versions (releases) is present and (e) at least one of its dependants (releases) is present.
- for change edges, (f) if a change edge (r,r') is present then both r and r' are present, and (g) conversely if two nodes r and r' connected by a change edge are present then the change edge is present.

Here by present we mean that some node or edge is kept in the solution, *i.e.*, the corresponding variable value is 1.

The set of linear constraints that encode these conditions is the following one, the correspondence being (1) ⇔ (a), (2)⇔ (b), (3)⇔ (c) ∧ (d), (4)⇔(e), and (5) ⇔ (f) ∧ (g).[1]

$$\mathrm{v}_{\mathrm{p}}^{\mathrm{rel}} = 1 \tag{1}$$

$$\forall d = (r, l) \in E_D, \mathrm{v}_{\mathrm{l}}^{\mathrm{lib}} \geq \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \tag{2}$$

$$\forall l \in N_L, \sum_{l_i \in \{l_i \mid (l, l_i) \in E_V\}} \mathrm{v}_{\mathrm{l_i}}^{\mathrm{rel}} = \mathrm{v}_{\mathrm{l}}^{\mathrm{lib}} \tag{3}$$

$$\forall l \in N_L, \sum_{r \in \{r \mid (r, l) \in E_D\}} \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \geq \mathrm{v}_{\mathrm{l}}^{\mathrm{lib}} \tag{4}$$

[1]A proof is available at [17].

$$\forall e = (r, r') \in E_C, \mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} = \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \times \mathrm{v}_{\mathrm{r'}}^{\mathrm{rel}} \tag{5}$$

which, using linearization, becomes $\forall e = (r, r') \in E_C$,

$$\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} \leq \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \tag{6a}$$

$$\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} \leq \mathrm{v}_{\mathrm{r'}}^{\mathrm{rel}} \tag{6b}$$

$$\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} \geq \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} + \mathrm{v}_{\mathrm{r'}}^{\mathrm{rel}} - 1 \tag{6c}$$

5.3 From MO-MC to SO-SC optimization

Now that we have defined what a correct update solution is, we would like to find one solution that is indeed optimizing conflicting criteria, *i.e.*, quality and cost. Multi-Objective Multi-Criteria Decision-Making is the field concerned with solving such problems [31]. The difficulty there stems from the presence of more than one criterion, with some to be maximized and some to be minimized. Multiple Pareto optimal solutions usually exist in such a case. Therefore, many methods to solve Multi-Objective Multi-Criteria (MO-MC) optimization problems proceed by transforming them into a Single-Objective Single-Criterion (SO-SC) problems.

From MC to SC. We use one of these methods, called Simple Additive Weighting (SAW) [10]. This method is based on *weights* assigned by the developer to each criterion. SAW consist in combining multiple criteria values into a single criterion value using a weighted sum. Before this phase, SAW requires a *normalization* phase to scale criteria values and be able to compare the ratings of all existing solutions. Some of the criteria are positive (wrt maximizing an objective function), *i.e.*, the higher the value, the higher the quality. This includes popularity metrics such as stars or downloads. Other criteria are negative (wrt maximizing an objective function), *i.e.*, the higher the value, the lower the quality. This includes criteria such as the cost of change or the vulnerability score related to CVEs that we use in our experiments. There are several normalization techniques [28]. To choose one of them, we need to take into account the objective function (should it be maximized or minimized) and the nature of the criteria (positive or negative wrt the objective function).

From MO to SO. The linear programming solver attempts either to maximize or minimize the value of the objective function by adjusting the values of the decision variables while enforcing the constraints. In dependency update, we are trying to maximize some quality metrics (*e.g.*, popularity) and to minimize the change cost and other quality metrics (*e.g.*, vulnerability), at the same time. To make this amenable to a SO problem we adopt the following viewpoint. We will consider that *quality metrics are a measure of a form of quality debt.* Hence to be minimized too. This way we only have criteria to minimize. This means, *e.g.*, that positive metrics such as popularity are to be taken as negative criteria in minimizing. Negative metrics such as CVEs based vulnerability or release age are, conversely, positive criteria in minimizing. This inversion is due to the fact that we use an objective function to be minimized and not maximized.

Normalization. We can now define our normalization process. Since we use an objective function to be minimized, values q_i for a positive metric q are scaled according to $\frac{q^{\max}-q_i}{q^{\max}-q^{\min}}$ where $q^{\min}$ and $q^{\max}$ are respectively the minimal and maximal possible values for q. For example, suppose that release popularity ranges from 10 stars to 100 stars. The normalized value for a release with 80 stars (which is quite good) is $\frac{100-80}{100-10} = 0.22$. Accordingly, values for a negative metric are scaled using $1 - \frac{q_i^{\max}-q_i}{q_i^{\max}-q_i^{\min}}$. For example, suppose that release age ranges between 5 days and 365 days. The normalized value for a release that is 300 days old (which is quite bad) is $1 - \frac{365-300}{365-5} = 0.82$.

In order to increase the solver efficiency [3] and to make the possible feed-back of quality debt enhancement more legible for developers, we apply a multiplicative factor k (set to 1000 by now) when normalizing, *e.g.*, for the examples above one would indeed get 220 and 820.

5.4 Optimization objective function

We must now establish the objective function of the linear program. This function will be minimized to identify the optimal solution for updating dependencies, referred to as *sol*. Our proposed objective function is outlined as follows:

$$Min\Big((1 - w(cost)) \times Quality_{sol} + w(cost) \times Cost_{sol}\Big) \qquad (7)$$

where, $w(cost)$ is the weight assigned by the developer to the cost of change metric, and $1 - w(cost)$ is the weight of the overall quality of the solution, referred to as $Quality_{sol}$. This overall quality is in turn computed by the use of the following aggregation function:

$$Quality_{sol} = \sum_{\forall q \in Q} w(q) \times f_q \qquad (8)$$

where $w(q)$ is the weight that the developer assigns to the quality metric q, and f_q is the aggregation function for the computation of the quality metric q of solution *sol*. The aggregation function of any metric is dependent on the nature of the criterion it seeks to aggregate. For instance, the aggregation function for vulnerability related to CVEs is defined by the sum of the vulnerability values of each release within the solution *sol*, as illustrated below:

$$f_q = \sum_{\forall r \in sol} q(r) \times \mathrm{v}_{\mathrm{r}}^{\mathrm{rel}} \qquad (9)$$

where $q(r)$ is the vulnerability value of release r and $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$ is used to prune releases that are not in the solution (remind that $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$ is 1 if release r is present in the solution, else it is 0).

Similarly, the cost of change metric values have to be aggregated. The cost of change for the solution *sol* is computed as the sum of the cost of change values associated with change edges within *sol* as follows:

$$Cost_{sol} = \sum_{\forall (r,r') \in sol} cost(r, r') \times \mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}} \qquad (10)$$

again, with $\mathrm{v}_{\mathrm{rr'}}^{\mathrm{chg}}$ being used to prune change edges that are not present in the solution.

5.5 Retrieval of the updated set of dependencies

Given the solution, the optimal set of (direct and indirect) dependencies correspond to all nodes r in N_R such that $\mathrm{v}_{\mathrm{r}}^{\mathrm{rel}}$=1 in the solution. Yet, a specificity of the maven package manager is that, whenever there are several paths from the project to some library l, it is the shortest path that is used to discriminate the version of l to be used at run time. Take for example l6 in Fig. 1. There are two paths from the root to it: one ending with l5-1, requiring version 2 and one ending with l2-2, requiring version 1. The later being the shortest. This means that if the best solution is (l1-3, l2-2, l5-1, l6-2) one cannot just update p's dependency file using l1-3 instead of l1-1, and l2-2 instead of l2-1, because at run-time l6-1 would be used and not l6-2. A simple solution to this is to update p's dependency file with all releases in the solution, here l1-3, l2-2, l5-1, and l6-2. On the one hand, this solution freezes things, but it ensures the quality that we announce.

6 EXPERIMENTAL EVALUATION

In this section, we describe the empirical evaluation of our approach, driven by the following research questions.

RQ1 (correctness): Does our approach generate correct updates? For zero-cost updates, Does our approach generate updates that compile and pass tests [8]?
RQ2 (effectiveness): Which quality gain and update cost can be expected by using our approach and how does this compare to naive approaches?
RQ3 (performance and scalability): How does our approach perform on different graph sizes and with different strategies?

In this evaluation, we compare our solution to three different naive solutions to update each of the direct dependencies, which are typically implemented in popular dependency management bots. Max version ("MMP") always picks the latest release, max version with same major ("mMP") picks the latest release within the same major version, and max version with same major and minor ("mmP") picks the latest release within the same major and minor versions. Our data and results are available at [17].

6.1 Subject application & methodology

Choice of metrics. The quality of releases can be measured by various metrics, each one representing some aspect to be taken into account in dependency update. Here, we introduce the quality metrics which we use in our experiments. We remind the reader that these are normalized before being used by the LP solver.

- *CVE (vulnerability score).* Our vulnerability score is based on the set of CVEs that (directly) impact a release. This set, denoted as C_{r} for a release r, is computed using Goblin. To obtain the score, we use the formula $\sum_{c \in C} k(c) \times nb(c, C_{\mathrm{r}})$ where $C = \{low, moderate, high, critical\}$ denotes the criticality of a CVE, $nb(c, C_{\mathrm{r}})$ counts the CVEs of criticality c in C_{r}, and k is a function associating a coefficient in the Fibonacci suite 2, 3, 5, and 8 to each element c in C.
- *Freshness (days between a release and the latest release).* Freshness [7] materializes the age of a release. Given some release r of a library l, freshness can be computed using the release date of r and some more recent date of reference, *e.g.*, the present day, the day ecosystem information was last extracted, or the release date of the last version of l. In our experiments we use the later, which is directly computed by Goblin.

Table 1: Configurations used for experiments

id	metrics			cost	strategy	
	F	P	CVE		for releases	for cost
cfg1	1.0	–	–	0.0	global	local
cfg2	1.0	–	–	0.5	global	local
cfg3	1.0	–	–	$\leq$ 0.0	global	local
cfg4	1.0	–	–	$\leq$ 0.0	local	global
cfg5	0.4	0.4	0.2	0.5	global	local
cfg6	0.4	0.4	0.2	0.5	local	local

- *Popularity (number of dependants over a 1-year window).* There are several possibilities for popularity too [32]: "stars" on social development forges, downloads, *etc.* The former requires all libraries in the ecosystem to be present on the social network, which is not the case. Further, stars apply to libraries, not releases. The number of downloads is sensitive to inflation issues [19] and to the passing of time, older releases being more downloaded by nature. An alternative to the first issue is to use the number of dependants; an alternative to the second one is to use a time window. Thus, in our experiments, we use the number of dependants in the ecosystem over a 1-year time window and have extended Goblin to take this metric into account.

Configurations. In our approach there is a high degree of variability regarding graph computation strategies and the weights one can assign to the quality metrics. In this evaluation, we choose to focus on six representative configurations depicted in Table 1.

In the "metrics" column, F stands for freshness and P for popularity, with associated weights given. Although our tool supports expressing an upper bound on the number of CVE vulnerabilities in the solution, we do not evaluate it here. Similarly, the cost metric allows expressing both weights (*e.g.*, 0.5 for cfg2) and hard constraints (*e.g.*, $\leq$ 0.0 in cfg3 which forces the cost of the solution to be zero). While weights and constraints can be mixed, we do not evaluate this scenario here. The last two columns are relative to the strategies used. For releases it means either developing alternatives to direct dependencies only (local) or for all libraries in the graph (global). For cost it means computing change arcs and costs either only at the root level (local) or anytime one has a library with more than one version (global). Costs are computed either using Maracas at the root level or Japicmp at any other level. Figure 1 is an example where a local strategy is used for both releases and costs. With a global strategy for releases, possibly more versions of l3–l6 would be present. With a global strategy for costs, there would be change edges between the two dependants of l6 and the two versions of this library (hence, four more change edges). In addition, all the configurations here use a strategy where only releases newer that the current ones are considered (*i.e.*, our approach will not recommend downgrading a dependency).

We use configurations 1 to 4 to compare our approach against naive approaches under various conditions for RQ1 and RQ2, exclusively employing freshness as quality metric. This aligns with naive approaches that solely aim to update dependencies to the latest possible versions. cfg1 does not account for cost and is thus the one closest to naive approaches. cfg5 and cfg6 are those that exploit the full potential of our approach which we use for RQ3.

Table 2: Demographics of our dataset of 107 Java projects

	Q1	Q2	Q3	min	max
p's direct dependencies	4.5	7	9	2	22
releases (N_R)	138	336	2771	16	39474
libraries (N_L)	12.5	40	134	4	960
dependency edges (E_D)	114.5	394	7066	9	141429
versions edges (E_V)	137	335	2770	15	39473

Dataset. To evaluate our approach, we reuse the dataset of 462 well-tested Java projects from Hejderup and Gousios [13]. Additionally, we filter projects that cannot be cloned (8), lack a root `pom.xml` file (13), or feature multi-module `pom.xml` hierarchies (216) which are not supported in our tool. We also filter projects that have not been updated since 2020 (77), that cannot be analyzed by Maracas (8), that have missing direct dependencies (14), or that take prohibitively long time to analyze (19). In the end, we obtain a subset of 107 well-tested and active Java Maven projects that can be analyzed with our tools. Table 2 details descriptive statistics of the resulting dataset in terms of rEDG size characteristics for cfg1 (different configurations yield different graph sizes).

Comparing solutions. Comparing solutions requires a common referential. The space of all possible solutions, *i.e.*, the rEDG and the normalized values for the different quality and cost metrics, play this part. As an illustration, Figure 1 contains both the solution we get with our approach and several solutions we may get with naive strategies such as using the more recent releases of the direct dependencies ({l1-3, l2-2, l5-1, l6-1} due to Maven's shortest path semantics) or using the more recent zero-cost releases ({l1-2, l2-1, l4-1, l5-1, l6-2} if we suppose that cost p/l1-1/l1-2 is zero and cost p/l1-1/l1-3 and cost p/l2-1/l2-2 are non-zero). To compute the (aggregated with weights) quality value, the cost value, and the global value of a solution is then just using the formulas $Quality_{sol}$, $Cost_{sol}$, and their weighted sum, as given in Section 5.4. This does not require to use the solver, only the rEDG and normalized values for its release nodes and change arcs.

Methodology. For RQ1 (correctness), we take each project that can be successfully compiled and tested in our environment before updating dependencies. For these projects, we launch our tool with configurations 1 to 4 to retrieve the proposed updates and compare them with those proposed by the three naive approaches (MMP, mMP, and mmP). We update the `pom.xml` file according to the solution before building and testing the project again to check whether compilation and tests are still successful.

For RQ2 (effectiveness), we compare the graph quality and cost obtained by each naive approach and ours. All the values given in this section are normalized on the same graph so that they can be compared.

Finally, for RQ3 (performance and scalability), we run our tool with cfg5 and cfg6 on all projects within our dataset. We retrieve for each execution various operational metrics such as execution times, graph size, memory usage, and the obtained solutions. Between each launch, we clean the Goblin database to avoid memoization and caching biases between subsequent runs.

Experimental setup and reproducibility. All the experiments presented in the rest of this section were carried out on a Windows Server 2019, 64 GB memory, 8 CPUs Intel(R) Xeon(R) CPU E7-8880 v4 @2.19GHz on a Docker instance from openjdk:17-jdk image (Oracle Linux 8). The Maven Central dependency graph is the version dated April 12, 2024 [14] and the CVE database is dated May 07, 2024 [17]. Experiments are based on version 1.0.0 of our Goblin-Updater tool [16]. We use version 2.1.0 of Goblin, version 0.5.0 of Maracas, version 0.17.2 of Japicmp, and version 9.8 of OR-tools.

6.2 Results

RQ1 (correctness). Among the 107 projects in our dataset, only 48% compiled and tested successfully in our environment without any intervention, yielding a total of 51 projects. We exclude two of these because cfg4, which calculates transitive costs, could not complete in two days. This highlights the substantial computational resources required to implement a full-cost approach. The analysis in this section considers the remaining 49 projects.

Figure 2 shows the percentage of projects that still compile and pass tests after the update suggested by naive approaches and different configurations of our approach. Note that, for cfg3, our approach did not find a zero-cost solution for 17 projects, so their solution graphs remain unchanged and the projects continue to compile and pass tests successfully. The same situation arises for 28 projects in configuration cfg4. We can see that the local zero-cost approach (cfg3) does not always yield a solution that compiles and passes the tests. This is because this configuration attempts to update all dependencies in the graph (not just direct dependencies like naive approaches) but only calculates cost on direct dependencies. To maximize the chances of finding a zero cost solution, we use cfg4 which combines Maracas to compute cost for direct dependencies and Japicmp to compute cost for indirect dependencies. With this zero-cost configuration, we achieve better results than any other approaches. However, it does not achieve 100% success in compiling and testing due to arbitrary constraints in the configuration of two projects that enforce specific versions and licenses to be used. It is also interesting to note that naive approaches that update direct dependencies to the latest release within the current major version or current major/minor version also yield projects that cannot be compiled or tested. This confirms that one cannot fully trust semantic versioning within the Maven ecosystem [23].

RQ2 effectiveness. The data presented in this section is derived from the same 49 projects discussed above, illustrating the correlations among compilation, testing, quality, and cost. cfg1, cfg2, and cfg3 share both identical graphs and metrics, thus ensuring uniform normalization in terms of quality and cost values. In contrast, cfg4 employs a different graph, and naive approaches lack quality and cost values. To be able to compare solutions, we have applied normalization to the cfg4 and the naive approaches' graphs using values from the graph of one of the earlier configurations, standardizing in this way the normalization process for all solutions.

Figure 3 shows the varying levels of freshness across solutions generated by different approaches. Since quality is assessed over

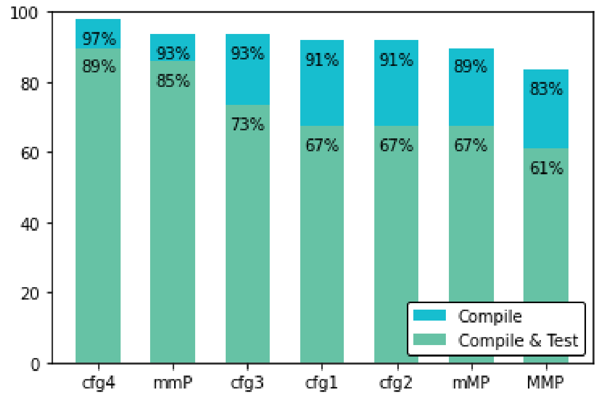


Figure 2: Correctness of the solutions generated by different approaches and configurations

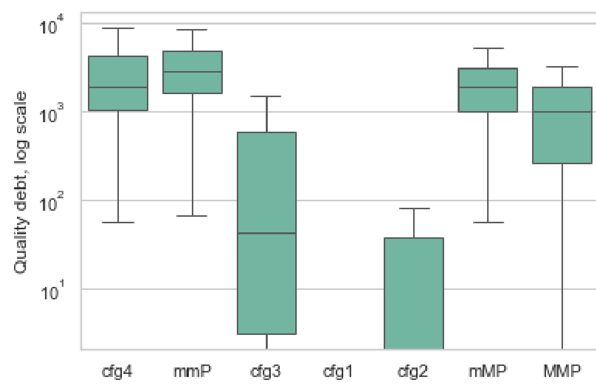


Figure 3: Freshness of the solutions generated by different approaches and configurations

the entire graph, even the naive approach of setting all direct dependencies to their most recent version fails to deliver optimal quality in the majority of cases. This highlights the importance of taking transitive dependencies into account when assessing the quality of software dependency graphs. Configuration cfg1 consistently yields solutions of maximum quality as it ignores cost constraints. This approach selects the most recent versions for all nodes in the graph, thereby ensuring the highest possible quality. Configuration cfg2, which maintains a quality/cost balance of 50%, provides solutions of average quality and cost. Finally, configurations cfg3 and cfg4 are looking for zero-cost solutions and are paying the price in terms of quality.

Figure 4 shows the varying costs associated to the solutions generated by different approaches. Here, the results are normalized based on the local cost metrics between root and direct dependencies. Figure 4 presents the aggregate costs for updating projects to fit with the new direct dependencies, as calculated by Maracas, across different approaches. Configurations cfg3 and cfg4 consistently provide solutions with the minimum cost because they constrain the solver to achieve zero-cost solutions. Configuration cfg1 gives the same cost as the naive MMP method, as the cost is only calculated

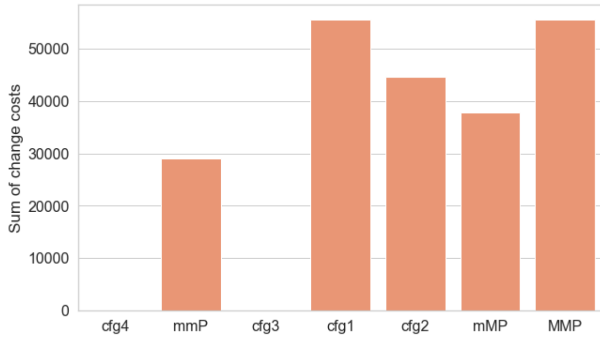


Figure 4: Cumulative cost of change of the solutions generated by different approaches and configurations

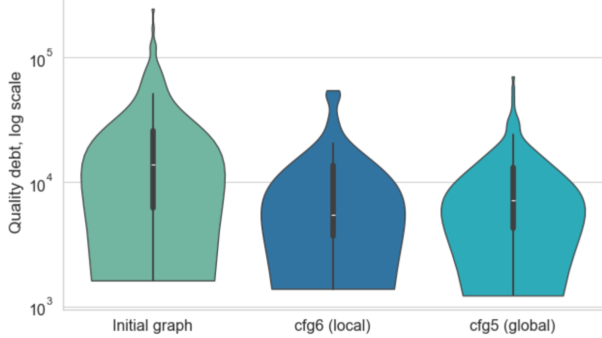
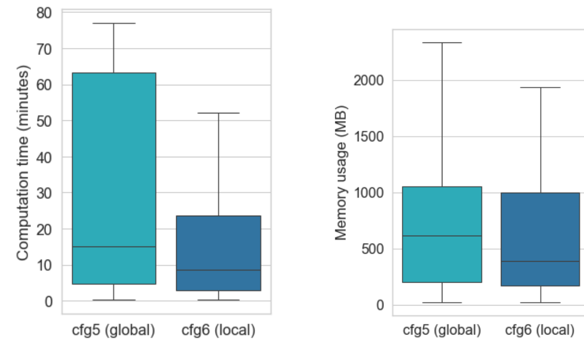


Figure 5: Quality debt before and after update

locally and the same direct dependencies are used, the point of variation being the transitive dependencies. Finally, the balanced configuration cfg2 yields update costs in between mMP and MMP, but delivers much better quality (Figure 3). Our different configurations deliver results in line with expectations regarding quality and cost. It is up to the maintainer to select the optimal configuration to meet specific requirements by adjusting the tool's parameters accordingly.

RQ3 performance and scalability. We start by comparing the difference in expected quality gain between a global (cfg5) and a local (cfg6) approach. As a reminder, the local approach considers all versions of direct dependencies, while the global approach considers all versions of (possible transitive) dependencies. The global approach therefore adds possibilities but increases the size of the solution graph. Figure 5 shows the distribution of initial project quality debt and that of a local and a global approach. Surprisingly, the difference in quality between the local and global approaches is small. The local approach delivers on average a reduction in quality debt of 57% and the global approach 60%.

Let us now compare the distribution of average execution time (Figure 6.a) and memory used (Figure 6.b) for these two approaches. Even if memory consumption does not vary significantly between the two approaches, the execution time differs greatly, going from an average of 19 min locally to 120 min globally (+533%) taking into account fliers. The global approach means much longer calculation



(a) Execution time **(b) Max memory usage**

Figure 6: Time and memory for updating projects

times for very little potential quality benefit. We therefore believe that the average user should most likely opt for the local approach, though the global approach may still be useful in certain scenarios.

Figure 7 dives deeper into different phases of our approach. In the global case, the generation of the graph and the solving phase dominate execution times. The graph generation includes the creation of the rDG and the weaving of metrics onto nodes and edges (realized using Goblin). In fact, the popularity metric consumes a lot of time as it requires to retrieve the associated library, to identify the dependants that use the appropriate version, and to retrieve these dependents to apply the 1-year time window. This whole process is extremely time-consuming for popular libraries. In normal circumstances the Goblin memoization would help greatly, but we desactivate it for our experiments. The computation time of our approach varies greatly with the chosen quality metrics and their individual computation times. Also, solving time represents a significant cost, and as we would expect, it will increase with the size of the graph and the number of constraints associated with the problem. The generation of change edges and the computation of costs is stable between the two configurations in terms of time, which means that it will take a large proportion when the calculation of the graph, metrics and solving are low and conversely if they are high.

Finally, we look at how to estimate the computation time of our approach on a project. In Figure 8.a, we can see that the number of direct dependencies does not have much impact on the total computation time. On the other hand, we can see from Figure 8.b that the total number of nodes is graphs does. This means that, unfortunately, computation times cannot be reliably estimated from the number of direct dependencies. To conclude, the execution times of our tool depend on the size of the graph, the number of constraints in the problem and the quality metrics chosen.

7 THREATS TO VALIDITY

For the construction of our threats to validity, we follow the structure proposed by Wohlin *et al.* [29].

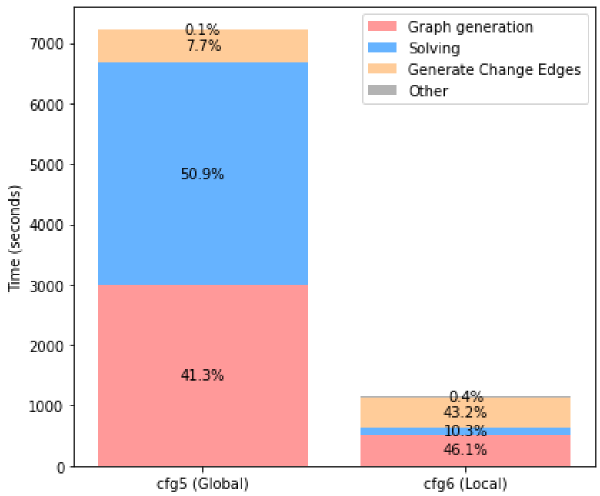
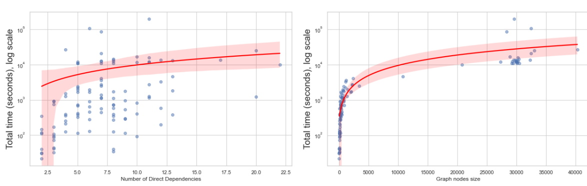


Figure 7: Execution time distribution



(a) Time (log) / direct dependencies **(b) Time (log) / graph nodes**

Figure 8: Relation between size and computation time

7.1 Internal & construct validity

In our approach, we operate under the hypothesis that the external tools we rely on (Goblin, Maracas, Japicmp, and OR-tools) yield accurate results and that the ecosystem dependency graph and metric values associated with release nodes, provided by Goblin, are both complete and accurate. This is reasonable as these tools have been independently evaluated by other researchers. As we used tests to check for update correctness, it should be noted that Hejderup and Gousios Hejderup et al. have shown that tests may have a low coverage of calls to dependencies (58% for calls to direct dependencies and 21% for calls to indirect ones [8, 13]). To mitigate this risk, we have used a dataset of well-tested projects. Furthermore, for calls to direct dependencies, we alleviated the issue with our cost of change computation using Maracas, which plays the role of the Updatera tool proposed in [13] for finding semantic breaks but goes further in being integrated *within* the update process itself. Yet, this still means that correctness computed with tests is a lower bound [8].

Our approach assumes that dependency versions are fixed, thereby excluding version ranges (*e.g.*, `[1.0, 2.0)`). In the Java/Maven context, this is not an issue, since the use of dependency ranges is almost non-existent. To verify this, we have examined the whole Maven Central dependency graph as of January 26, 2024. It revealed that only 1.12% of dependencies (*i.e.*, 1,173,629 dependencies out of 104,949,615) employ ranges. Our tool uses a POM file to infer the list of direct dependencies of a project. The adaptation to other file formats (*e.g.*, Gradle's build files) should be straightforward. At present, the cost of change is based on counting the number of breaking uses identified by Maracas and/or Japicmp. This can be simplistic, as one can easily imagine that the cost associated with a renamed method differs from that of a removed method, or that n identical changes yield a lesser cost compared to n distinct changes. Enhancing the cost function is a priority for future evolution perspectives. Finally, for the experimental evaluation part, we have discarded certain projects from the study that were prohibitively long to analyze, thus reducing average calculation times.

7.2 External validity

Our approach does not account for libraries that are not hosted on Maven Central, such as those available through GitHub Package or proprietary dependencies internal to a company. In practice, Goblin could be extended for this, provided specific miners are developed. While our approach is adaptable to different software ecosystems, the primary concerns regarding generalization stem from the Java/Maven specifications. First, in contrast with Maven, ecosystems such as npm allow for multiple versions of a library to coexist in a project. The LP encoding can be modified for this, relaxing the single library version constraint. The absence of consideration for version ranges may be a challenge when applying our tool to ecosystems like npm where version ranges are used extensively.

8 DISCUSSION

In this section we present lessons learned in the use of our GoblinUpdater tool. Our approach and tool have different degrees of freedom:

- DF1: the strategy to retrieve an rDG from a project p, *i.e.*, where to compute sets of candidate versions (locally at the root direct dependencies or globally) and how to compute them (all versions, more recent versions),
- DF2: the strategy to obtain an rEDG from an rDG, *i.e.*, where to compute change edges (locally at the root direct dependencies or globally) and how to estimate cost (Maracas and/or Japicmp),
- DF3: where to compute the quality metrics (on the Goblin side, on the GoblinUpdater side, or both),
- DF4: the chosen set of quality metrics, associated weights, and possible hard constraints for CVE and cost.

We observed that global and version-exhaustive strategies for DF1 are costly, while using a local strategy and limiting the set of versions of a library to the ones newer than the current ones is more efficient. This is quite realistic considering the use one would make of our tool, a notable counter-example being, *e.g.*, the recent `xz` backdoor (CVE-2024-3094) which required downgrading from versions 5.6.0/5.6.1 to an earlier one.

The number of change edges selected by DF2 is also an important parameter. The computation of change edges is quite efficient. Yet, for each of them a JAR file must be retrieved (which can be done once, and offline) and Maracas or Japicmp has to check for all possible breaking changes, which is costly.

Metrics can be computed, following DF3, either on the Goblin side, on the GoblinUpdater side, or a combination of both. The first solution requires forking and extending the Goblin tool, but then the memoization mechanisms available in Goblin speed up the computation of metrics on releases that have already been analyzed. This is the solution we have used for our popularity metric. The second solution is easier but does not benefit from Goblin's Neo4J-based algorithms nor from memoization. The last one, where basic "atoms" are computed by Goblin and metrics are computed on top of these provides a good balance.

Finally, DF4 enables a personalized update solution with selected metrics and their respective weights chosen either at the organization or developer level. This choice is also connected to DF3: if a particular metric is of interest, one could incorporate it in Goblin to benefit from memoization.

9 CONCLUSION

Updating dependencies is a crucial practice in software development, especially for projects reliant on external libraries. While some developers might resist updating their project's dependencies to avoid the cost of change, others might prioritize finding a middle ground between the immediate costs of updating and the long-term benefits of maintaining up-to-date dependencies. This paper advocates for a balance between the benefits of maintaining up-to-date dependencies and the costs of updating. We introduce a multi-objective optimization approach to the dependency update dilemma, aiming to identify the most beneficial update solution based on criteria such as popularity, freshness, vulnerability and the minimization of breaking changes. We implement our optimization approach in a new tool GoblinUpdater, available at [16], and show using a dataset of 107 well-tested open-source Maven projects that it successfully finds update solutions and outperforms approaches typically implemented in dependability bots.

REFERENCES

[1] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2013. The Evolution of Project Inter-dependencies in a Software Ecosystem: The Case of Apache. In *29th International Conference on Software Maintenance (ICSM)*. 280–289.

[2] Amine Benelallam, Nicolas Harrand, César Soto-Valero, Benoit Baudry, and Olivier Barais. 2019. The maven dependency graph: a temporal graph-based representation of maven central. In *16th International Conference on Mining Software Repositories (MSR)*. 344–348.

[3] Timo Berthold and Gregor Hendel. 2021. Learning to scale mixed-integer programs. In *35th AAAI Conference on Artificial Intelligence*, Vol. 35. 3661–3668.

[4] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 255–265.

[5] Bodin Chinthanet, Raula Gaikovina Kula, Shane McIntosh, Takashi Ishio, Akinori Ihara, and Kenichi Matsumoto. 2021. Lags in the release, adoption, and propagation of npm vulnerability fixes. *Empirical Software Engineering* 26, 3 (2021), 1–28.

[6] Vašek Chvátal. 1983. *Linear programming*. Macmillan.

[7] Joel Cox, Eric Bouwers, Marko C. J. D. van Eekelen, and Joost Visser. 2015. Measuring Dependency Freshness in Software Systems. In *37th International Conference on Software Engineering (ICSE)*. 109–118.

[8] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely Updating Outdated Dependencies. In *45th International Conference on Software Engineering (ICSE)*. 233–244.

[9] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[10] Matthias Ehrgott. 2005. *Multicriteria optimization*. Vol. 491. Springer Science & Business Media.

[11] Jesus M. Gonzalez-Barahona, Paul Sherwood, Gregorio Robles, and Daniel Izquierdo. 2017. Technical Lag in Software Compilations: Measuring How Outdated a Software Deployment Is. In *Open Source Systems: Towards Robust Practices*. 182–192.

[12] Runzhi He, Hao He, Yuxia Zhang, and Minghui Zhou. 2023. Automating dependency updates in practice: An exploratory study on github dependabot. *IEEE Transactions on Software Engineering* (2023).

[13] Joseph Hejderup and Georgios Gousios. 2022. Can we trust tests to automate dependency updates? A case study of Java Projects. *Journal of Systems and Software* 183 (2022), 111097.

[14] Damien Jaime. 2024. Goblin: Neo4J Maven Central dependency graph. https://doi.org/10.5281/zenodo.10605656

[15] Damien Jaime, Joyce El Haddad, and Pascal Poizat. 2024. Goblin: A Framework for Enriching and Querying the Maven Central Dependency Graph. In *21st International Conference on Mining Software Repositories (MSR)*.

[16] Damien Jaime, Joyce El Haddad, Pascal Poizat, and Thomas Degueule. 2024. *Dupdater code*. https://github.com/Goblin-Ecosystem/goblinUpdater

[17] Damien Jaime, Joyce El Haddad, Pascal Poizat, and Thomas Degueule. 2024. Experiments data. https://zenodo.org/records/13285362

[18] Dhanushka Jayasuriya, Valerio Terragni, Jens Dietrich, Samuel Ou, and Kelly Blincoe. 2023. Understanding Breaking Changes in the Wild. In *32nd International Symposium on Software Testing and Analysis (ISSTA)*. 1433–1444.

[19] Jeremy Katz. 2018. Don't believe the download numbers when evaluating open source projects. https://blog.tidelift.com/dont-believe-the-download-numbers-when-evaluating-open-source-projects [Accessed: 2024-04-09].

[20] Amir M. Mir, Mehdi Keshani, and Sebastian Proksch. 2023. On the Effect of Transitivity and Granularity on Vulnerability Propagation in the Maven Ecosystem. In *30th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 201–211.

[21] Martin Mois. 2024. japicmp. https://siom79.github.io/japicmp/ [Accessed: 2024-04-09].

[22] Inc. Neo4j. 2024. Neo4j Cypher Query Language. https://neo4j.com/product/cypher-graph-query-language/ [Accessed: 2024-04-09].

[23] Lina Ochoa, Thomas Degueule, Jean-Rémy Falleri, and Jurgen Vinju. 2022. Breaking Bad? Semantic Versioning and Impact of Breaking Changes in Maven Central. *Empirical Software Engineering* 27, 3 (2022), 1–42.

[24] Renovate. 2024. Renovate: Universal dependency automation tool. https://github.com/renovatebot/renovate.

[25] Benjamin Rombaut, Filipe R Cogo, Bram Adams, and Ahmed E Hassan. 2023. There's no such thing as a free lunch: Lessons learned from exploring the overhead introduced by the greenkeeper dependency bot in npm. *ACM Transactions on Software Engineering and Methodology* 32, 1 (2023), 1–40.

[26] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the maven ecosystem. *Empirical Software Engineering* 26, 3 (2021), 45.

[27] Jacob Stringer, Amjed Tahir, Kelly Blincoe, and Jens Dietrich. 2020. Technical Lag of Dependencies in Major Package Managers. In *27th Asia-Pacific Software Engineering Conference (APSEC)*. 228–237.

[28] Nazanin Vafaei, Rita A Ribeiro, and Luis M Camarinha-Matos. 2016. Normalization techniques for multi-criteria decision making: analytical hierarchy process case study. In *7th advanced doctoral conference on computing, electrical and industrial systems (DoCEIS)*. 261–269.

[29] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering*. Springer Science & Business Media.

[30] Yuhao Wu, Yuki Manabe, Tetsuya Kanda, Daniel M German, and Katsuro Inoue. 2017. Analysis of license inconsistency in large collections of open source projects. *Empirical Software Engineering* 22 (2017), 1194–1222.

[31] K Paul Yoon and Ching-Lai Hwang. 1995. *Multiple attribute decision making: an introduction*. Sage publications.

[32] Ahmed Zerouali, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. 2019. On the Diversity of Software Package Popularity Metrics: An Empirical Study of npm. In *26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 589–593.

[33] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has my release disobeyed semantic versioning? static detection based on semantic differencing. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1–12.

[34] Hao Zhong and Na Meng. 2024. Compiler-directed Migrating API Callsite of Client Code. In *46th International Conference on Software Engineering (ICSE)*.